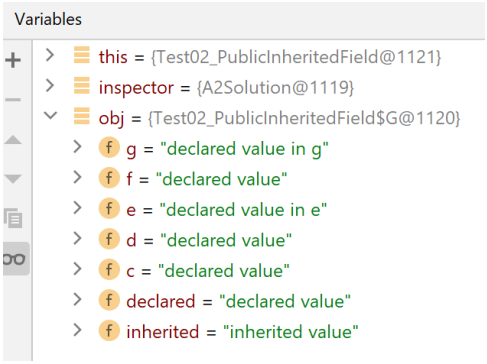
ASSIGNMENT 2

CS 474: Object-Oriented Languages and Environments / Fall 2021



Description

In this assignment, you will have to write a tool to inspect and modify all the fields belonging to a Java object. Your tool could be used to implement an IDE utility, as depicted above. You will use Java reflection to implement this assignment.

For this assignment, you need to write a class that implements the interface **`ObjectInspector`**, as shown below. You will be provided with startup code that you cannot modify (except as described in this document), and you only have to submit an implementation to the interface described below.

```
interface ObjectInspector {
    Map<String, String> describeObject (Object o);
    void updateObject(Object o, Map<String, Object> fields);
}
```

Method **`describeObject`** takes a single argument, which is the object owning the fields to describe. It returns a **map** with one entry for each field. The **key** is the name of the field. The **value** is the description of the field as follows:

- Primitive types
  - `int` fields are described as the result of `Integer.toString(int)`
  - `char` and `boolean` fields are similar to `int` but using methods `Character.toString(char)` and `Boolean.toString(boolean)`, respectively
  - `long` fields are described as the result of `Long.toString(long)` followed by the string "`#L`". E.g., `872349234#L`

    - `float` and `double` fields are similar to `long`, but described with `Float.toString(float)` and `Double.toString(double)` and followed by the string "`#F`" and "`#D`", respectively
    - `byte` fields are described as a string starting with "`0x`" and followed by the hex value of the byte, obtained through `Integer.toHexString(int)`. E.g., `0xA1`
    - `short` fields are described as a string starting with zero "`0`" and followed by the octal representation of the number through `Integer.toOctalString(int)`
- Boxed types
    - All boxed types are described with a string starting with "`Boxed `" and followed by the primitive description of the boxed value (see above)
- Null references are described as the string "`null`"
- All other objects are described by their "`toString`" method or by a method named "`debug`" if there is one (see below)

Method **`updateObject`** takes a map of names of fields to be updated as the key and the new value to write to each field as the value, and the object to perform those writes to. It should write the provided value to the provided field in the object. For instance, calling `updateObject(o, { "a": 0 , "b": 1})` should write the value `0` to field `o.a` and the value `1` to field `o.b`.

Static Fields

Static fields should be indicated by prepending the name of the class to the name of the field as the key to that field's entry. For instance, if class `A` has a static field `foo`, the key to that field should be "`A.foo`"

When the argument of **`describeObject`** is an instance of `java.lang.Class`, then the method should describe the **only** static fields belonging to the class represented by the argument. For instance, using the example in the paragraph above, `describeObject(Class.forName(A))` should return a map with a key "`A.foo`"

Exceptions

Invoking method `toString` and `debug` may result in an exception. In that case, the description of that field should be as follows:

- For errors, "`Raised error: `" followed by the fully qualified name of the class of the error. For instance "`Raised error: java.lang.Error`"
- For checked exceptions, "`Thrown checked exception: `" followed by the fully qualified name of the class of the exception. For instance, "`Thrown checked exception: java.io.IOException`"
- For unchecked exceptions, "`Thrown exception: `" followed by the fully qualified name of the class of the exception. For instance, "`Thrown exception: java.lang.NullPointerException`"

Debug Method

Instead of calling method `toString`, your code should call method `debug` if one exists, as described below:

- There is a non-static method called `debug` that returns a `String` and does not take any arguments.
    - Your solution should call this method with the object as the receiver
- There is a static method called `debug` that returns a `String` and takes a single argument.
    - You should pass the object being described as the single argument
    - If there are many such static methods, your solution can call any method that takes an argument compatible with the type of the object being described

~~Collisions~~

~~If there are many fields with the same name in different classes, your solution should disambiguate each field by using the following rules:~~

- ~~Static fields: Prepend the name of the class that defines the static field to the name of the field (e.g., "`A.field`" and "`B.field`")~~
- ~~Instance fields: Prepend the name of the class that defines the field with the string "`.this.`" (e.g., "`A.this.field`" and "`B.this.field`")~~

Entry Point

You should create a new class, on a new file, where you will implement your solution. You should change method Main.getExplainer so that it creates an instance of the class you added. You cannot change any other part of the code that is provided to you.

```
public abstract class Main {
    static ObjectInspector getInspector() {
        throw new Error("Not implemented");
    }
}
```

Due Date and Resubmission Policy

This assignment is due on **October 2 2021** (Saturday) at **5pm CST**. There is no late policy.

The code and date used for your submission is defined by the last commit to your Git repository.

To resubmit this assignment, your **original grade** (as defined by the autograder) should be **equal to or higher than 30%**. You can resubmit your assignment until **October 9 2021** (following Saturday) at **5pm CST**. Together with your resubmission, you will have to submit a written description of what you changed from the original submission (on Gradescope).

Bonus Points

This assignment has a total of **10% bonus points**, which you can earn by using Piazza as described in the syllabus. Your posts should be public, tagged with the `assignment2` label, and non-anonymous to the instructors to count towards the bonus.

Submission and Grading

This assignment is submitted through Github, and has an automatic grade component of 100%. You can check your current grade at any point by submitting your code and checking the autograder. The automatic grade is determined by 10 tests, that will check if your project outputs the expected result. Each test is worth 10%.

Graduate students should also submit a video explaining their solution. For graduate students, each test is worth 9%, for a total of 90%, and the video is worth 10%. **The maximum length for the video is 5 minutes.** This video should be a screencast of their IDE open on the code submitted, and the student should highlight the code and narrate the purpose of the highlighted code. You can record such a video without installing any software by using the following website: https://screenapp.io/#/

The grading rubrics for the screencast are as follows:

- Staying within the time limit
- Breaking down the problem into small methods that prevent/avoid copy-pasting code, and only showing the interesting methods directly related with this assignment
- Clear understanding of the reflection APIs used (e.g., getFields vs getDeclaredFields)
- Correct handling of exceptions

The final grade for the assignment will be the grade of the original submission, for assignments without a resubmission; or the average between the original grade and the resubmission grade, for assignments with a resubmission. The grade of the original submission includes any bonus points.

Errors and Omissions

If you find an error or an omission, please post it on Piazza as soon as you find it.

Hardcoding and Academic Integrity

Any hardcoding will result in a 0% grade. Hardcoding is when you submit code that detects which test is being run, and simply outputs the expected result. For instance, detecting that test 22 is running, and replacing the usual execution of your submission with `System.out.println("expected result").`

The academic integrity policy described in the syllabus applies to this assignment. You are responsible for writing all the code that you submit. We will use an automatic tool that detects plagiarism on all submitted code, and we will investigate all instances where plagiarism is more than likely.

Please refer to the syllabus for the full academic integrity policy.